5. Architectural Glue Patterns

Prof. Dr. U. Aßmann

Chair for Software Engineering

Faculty of Computer Science

Dresden University of Technology

11-1.0, 11/8/11

1) Mismatch Problems
2) Adapter Pattern
3) Facade
4) Some variants of Adapter
5) Adapter Layers
6) Mediator
7) Repository Connector

Literature (To Be Read)

▲ D. Garlan, R. Allen, J. Ockerbloom. Architectural mismatch – or why it is so hard to build systems out of existing parts. Int. Conf. On Software Engineering (ICSE 95)

http://citeseer.nj.nec.com/garland95architectural.html

▲ GOF – Adapter, Mediator

▲ Non-mandatory:

▪ Mirko Stölzel. Entwurf und Implementierung der Integration des Dresden OCL Toolkit in Fujaba. Großer Beleg. 2005. Technische Universität Dresden, Fakultät Informatik, Lehrstuhl für Softwaretechnologie
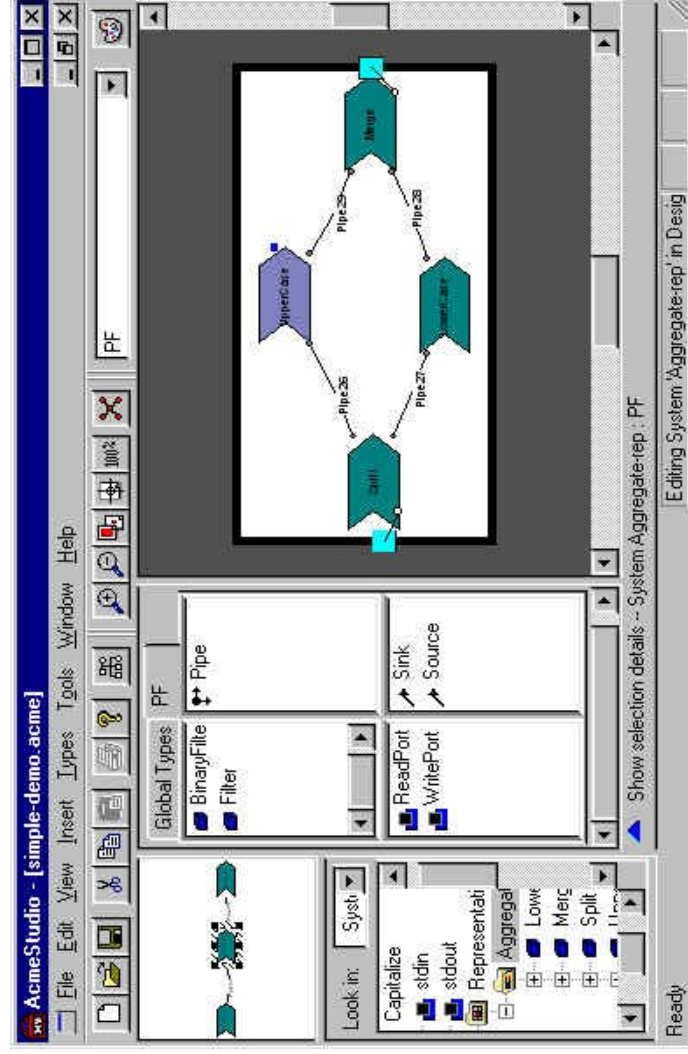
Goal

- Understand architectural mismatch
- Understand design patterns that bridge architectural mismatch

Architectural Mismatch

- Case study of Garlan, Allen, Ockerbloom 1995
- Building the architectural system Aesop

Architectural Mismatch

- ▲ Aesop was built out of 4 off-the-shelf components
  - OBST: an object-oriented C++ database
  - Interviews and Uniframe, a windowing toolkit
  - Softbench, an event bus (event-based mediator)
  - RPC interface generator of Mach (MIG)
- ▲ All subsystems written in C++ or C
- ▲ First Aesop version took 5 person years, and was still sluggish, very large
- ▲ Problems can be characterized in terms of components and connections

Interviews/Uniframe	Softbench
	MIG
	OBST
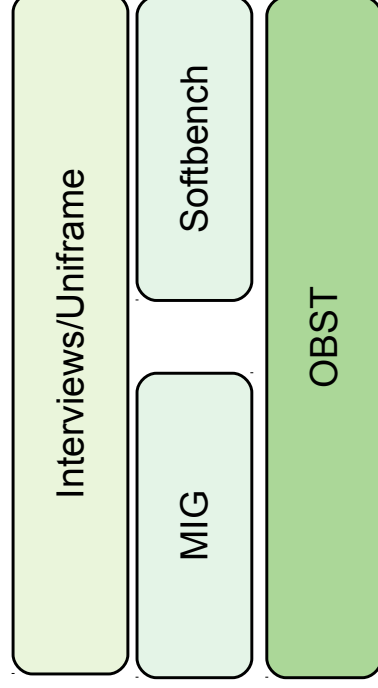
Classification of
Different Assumptions of the COTS

- ▲ Different Assumptions about the *component model*
  - Infrastructure
  - Control model
  - Data model
- ▲ Different assumptions about the *connectors*
  - Protocols
  - Data models
- ▲ Different assumptions about the *global architectural structure*
- ▲ Different assumptions about the *construction process*

component model	
	construction process
global architecture	
connectors	
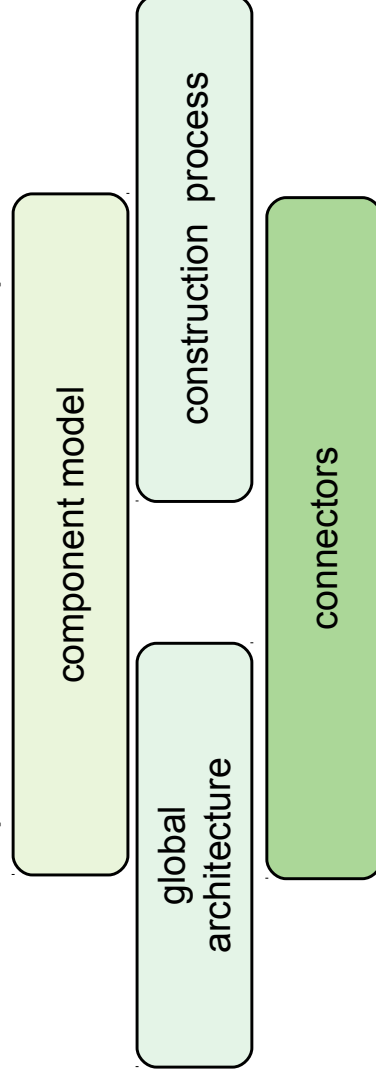
Different Assumptions about the Component Model

▲ A component model assembles information and constraints about the nature of components
- Nature of interfaces
- Substitutability of components

▲ Here: Infrastructure, Control model, Data model

▲ Different Assumptions about the Component Infrastructure:
- Components assume that they should provide a certain infrastructure, which the application does not need
- OBST provides many library functions for application classes; Aesop needed only a fraction of those

▲ Components assume they have a certain infrastructure, but it is not available
- Softbench assumed that all other components have access to an X window server (for communication)

▲ More in "Component-Based Software Engineering", summer semester

---

Assumptions on Control Model

▲ COTS think differently in which components have the main control
- Softbench, Interviews, and MIG have an ever-running event loop inside
- They call applications with callbacks (observer pattern)

▲ However, they use different event loops:
- Softbench uses X window event loop
- MIG and Interviews have their own ones
- The event loops had to be reengineered, to fit to each other

Assumptions on Data Model

- Different assumptions about the data
  - ▲ Uniframe: hierarchical data model
    - ■ Manipulations only on a parent, never on a child
    - ■ However, the application needed that
   - ■ Decision: rebuild the data model from scratch, is cheaper than modification

Assumptions about the Connectors

Protocol Mismatch

- Softbench works asynchronously; which superimposes concurrency to tools
  - Softbench is a mediator between tools
- 2 kinds of interaction protocols
  - Request/Reply (callback, observer): tool requests a service, registers a callback routine, is called back by Softbench
  - Notify via Softbench

Protocol Mismatch

- Softbench works asynchronously; which superimposes concurrency to tools, when messages of different tools are crossing

Tool 1 — Tool 2 — Softbench — Tool 3

Request A · Notify B · Reply A · Concurrency · Notify B
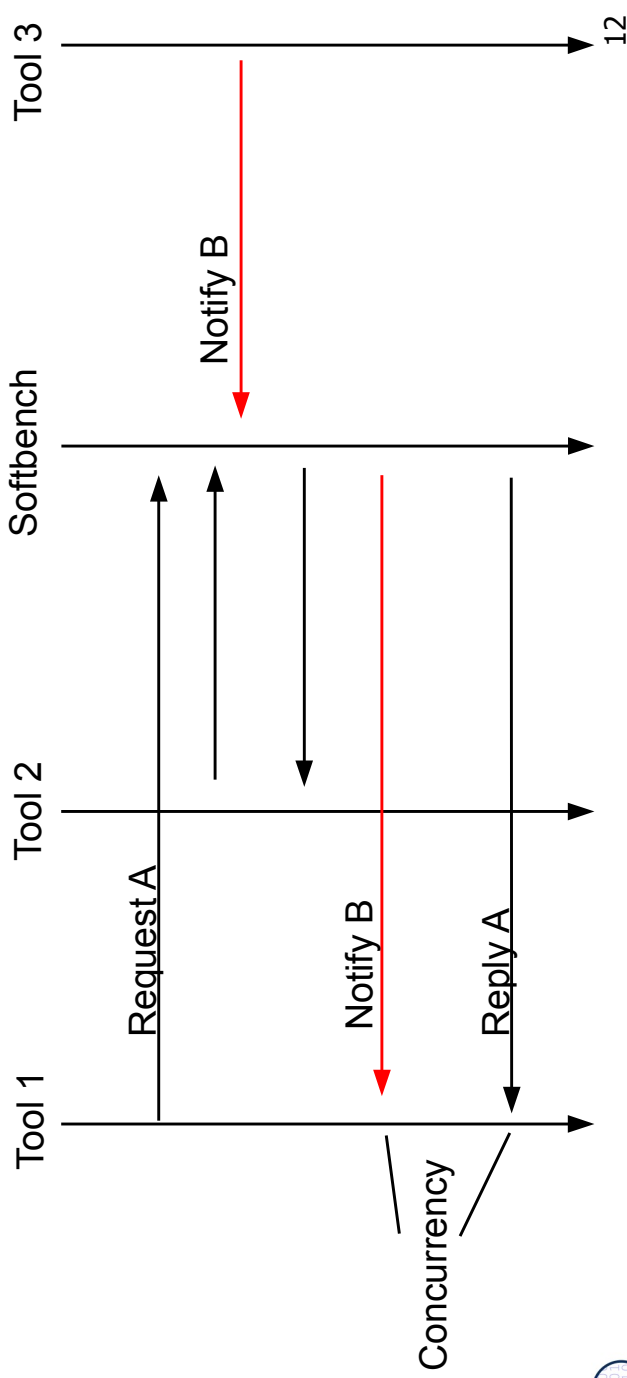
Data Format Mismatch

▲ Components also have different assumptions what comes over a channel (a connection).

- Softbench: Strings
- MIG: C data
- OBST: C++ data

▲ Requires translation components

- When accessing OBST, data must be translated all the time
- This became a performance bottleneck

Assumptions about the Global Architecture

▲ OBST

- Assumes a database-centered architecture
- Assumes independence of client tools
- And provides a transaction protocol per single tool, not per combination of tools
- Doesn't help when tools have interactions

Assumptions about the Building Process

- ▲ Assumptions about the library infrastructure
- ▲ Assumptions about a generic language (C++)
- ▲ Assumptions about a tool specific language
- ▲ Combination is fatal:
  - Some component A may have other expectations on the generated code of another component B as B itself
  - Then, the developer has to patch the generated code of A with patch scripts (another translation component)

Proposed Solutions of [Garlan]

- ▲ Make *all* architectural assumptions explicit
  - Problem: how to document or specify them?
  - Many of the aforementioned problems are not formalized
  - Implicit assumptions are a violation of the information hiding principle, and hamper variability
- ▲ Make components more independent of each other
- ▲ Provide bridging technology
  - For building language translation components (compiler construction, compiler generators, XML technology)
- ▲ Distinguish architectural styles (architectural patterns) explicitly
  - Distinguish connectors explicitly
- ▲ Solution: design patterns serve all of these purposes

Usability of Extensibility Patterns

- All extensibility patterns can be used to treat architectural mismatch
- Behavior adaptation
  - ChainOfResponsibility as filter for objects, to adapt behavior
  - Proxy for translation between data formats
  - Observer for additional behavior extension, listening to the events of the subject
  - Visitor for extension of a data structure hierarchy with new algorithms
- Bridging data mismatch
  - Decorator for wrapping, to adapt behavior, and to bridge data mismatch, not for protocol mismatch
  - Bridge for factoring designs on different platforms (making abstraction and implementation components independent)
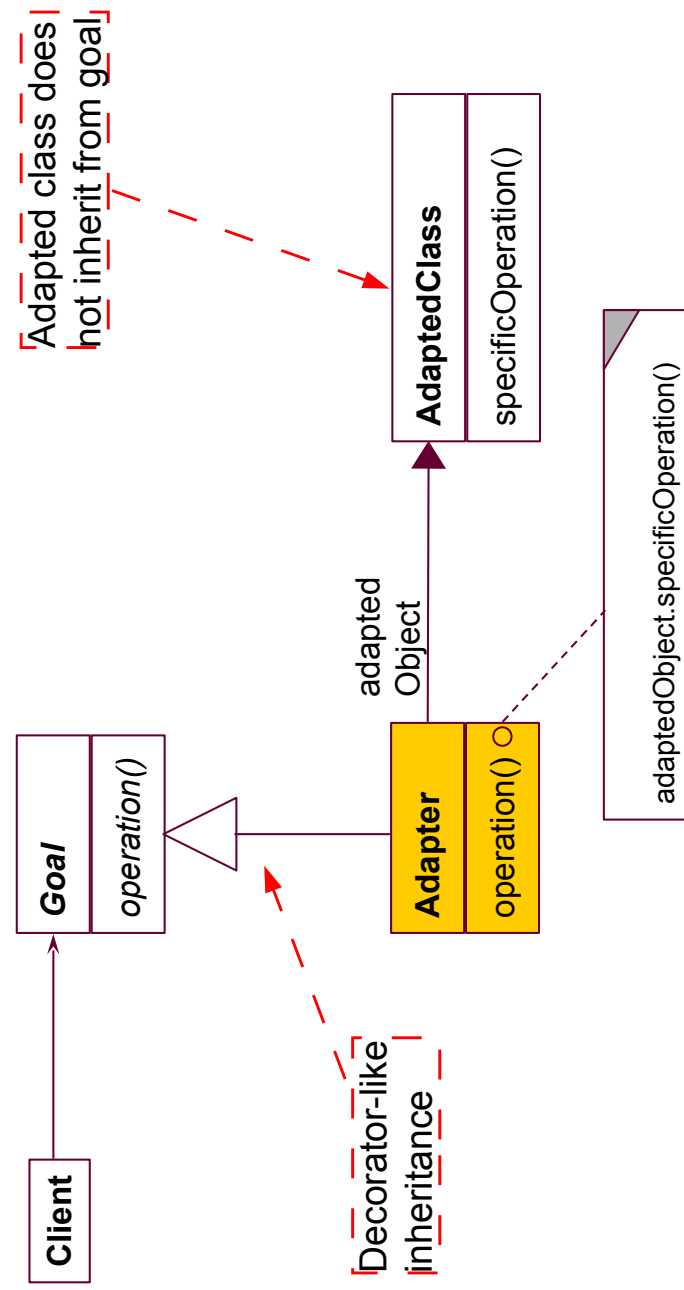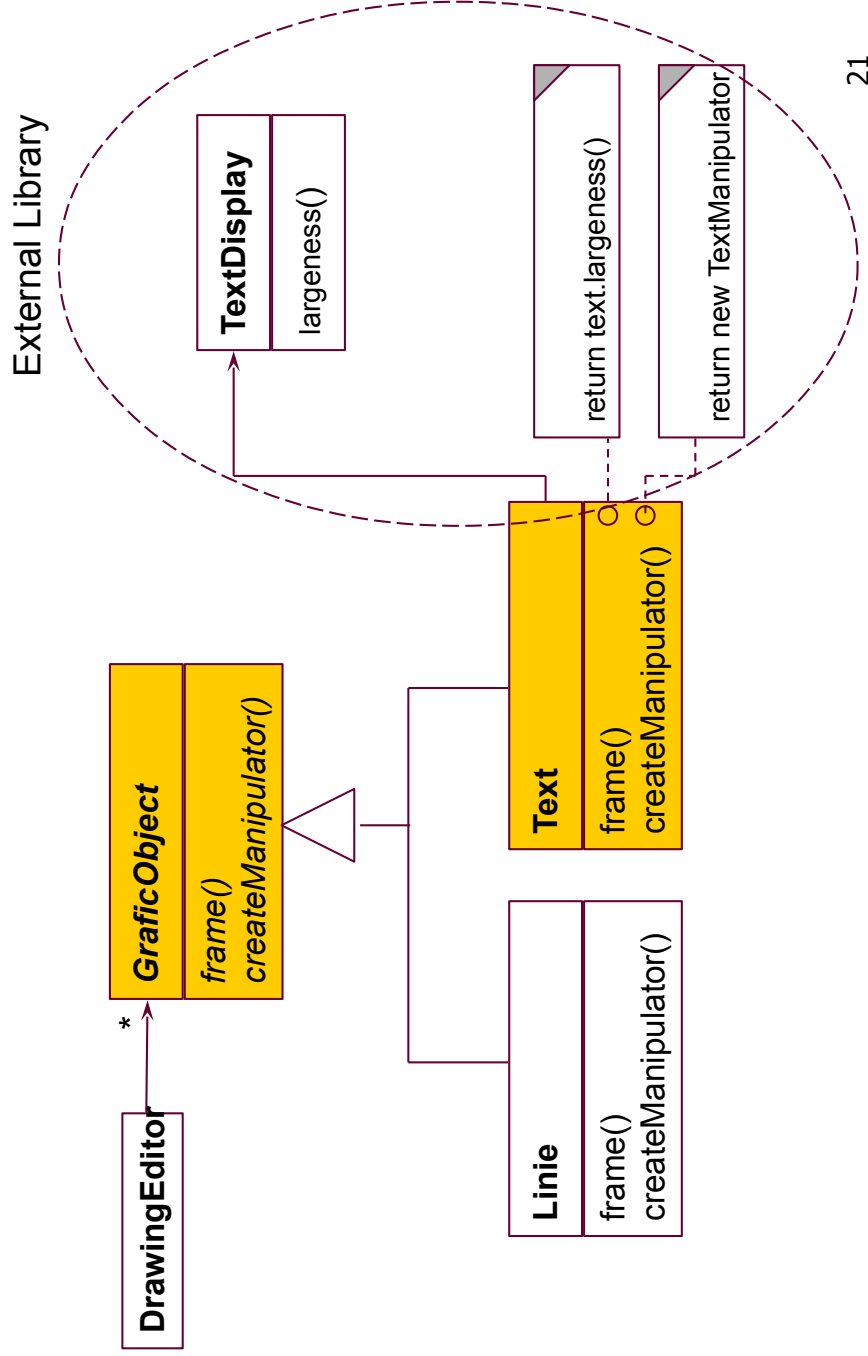
5.2 Adapter

Object Adapter

▲ An object adapter is a proxy that maps one interface to another
- Or a protocol
- Or a data format

▲ An adapter cannot easily map control flow to each other
- Since it is passed *once* when entering the adapted class

Object Adapter

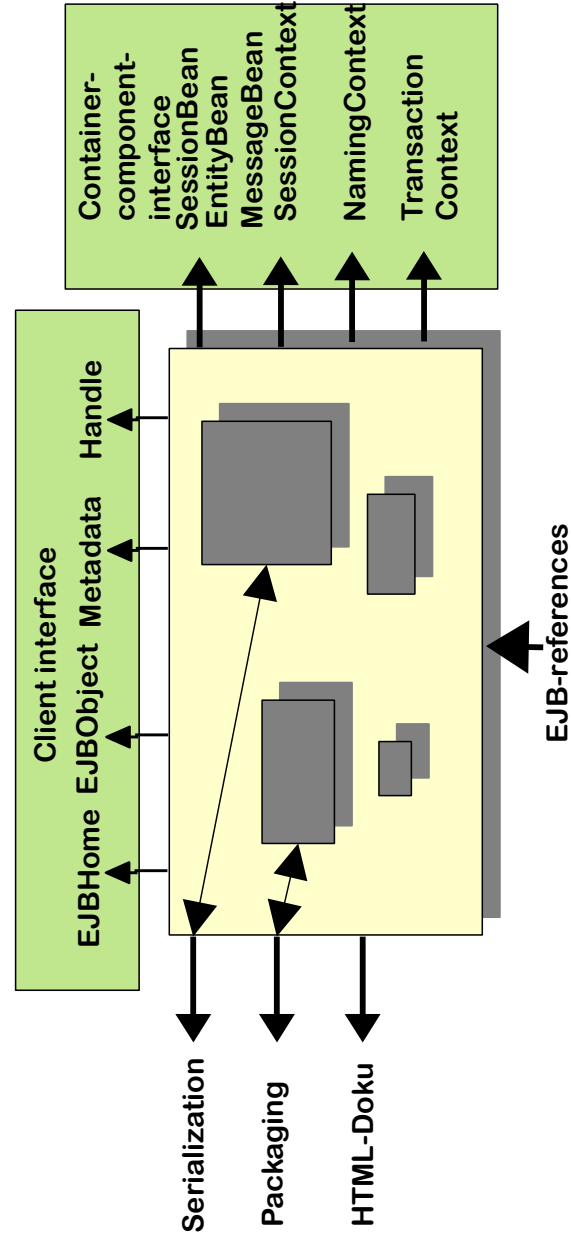▲ Object adapters use delegation

Adapted class does not inherit from goal

AdaptedClass
specificOperation()

adapted Object

adaptedObject.specificOperation()

Goal
operation()

Adapter
operation()

Client

Decorator-like inheritance

Example: Use of an External Class Library For Texts

External Library

DrawingEditor

*

GraficObject

frame()
createManipulator()

Linie

frame()
createManipulator()

Text

frame()
createManipulator()

TextDisplay

largeness()

return text.largeness()

return new TextManipulator
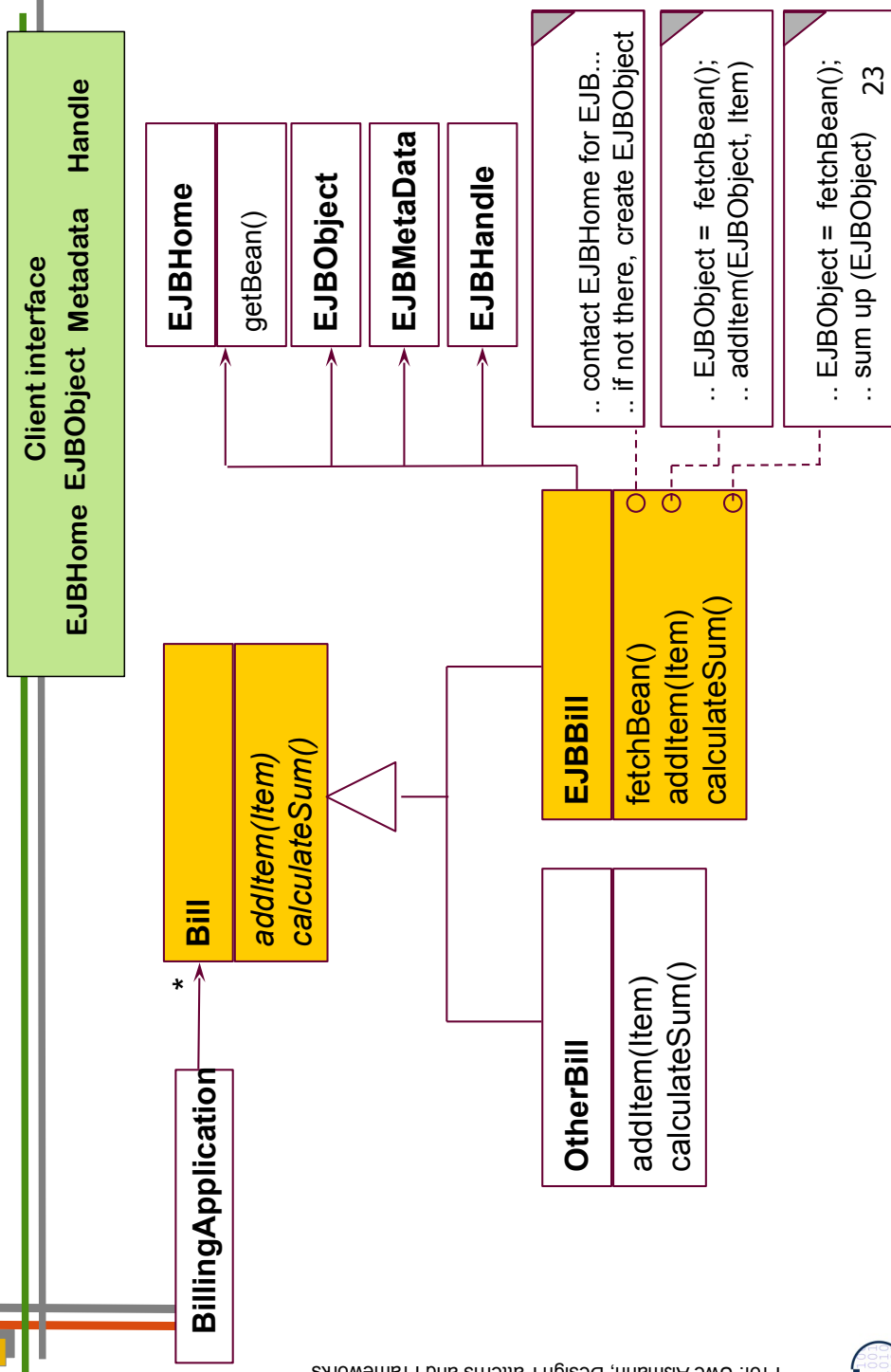
Prof. Uwe Aßmann, Design Patterns and Frameworks

21

Adapters for COTS

▲ Adapters are often used to adapt components-off-the-shelf (COTS) to applications

▲ For instance, an EJB-adapter allows for reuse of an Enterprise Java Bean in an application

Client interface

EJBHome EJBObject Metadata Handle

Container-
component-
interface
SessionBean
EntityBean
MessageBean
SessionContext

NamingContext

Transaction
Context

EJB-references

Serialization

Packaging

HTML-Doku

Prof. Uwe Aßmann, Design Patterns and Frameworks

22

EJB Adapter

Client interface
EJBHome EJBObject Metadata Handle

EJBHome
getBean()

EJBObject

EJBMetaData

EJBHandle

Bill
addItem(Item)
calculateSum()

*

BillingApplication

OtherBill
addItem(Item)
calculateSum()

EJBBill
fetchBean()
addItem(Item)
calculateSum()

.. contact EJBHome for EJB...
.. if not there, create EJBObject

.. EJBObject = fetchBean();
.. addItem(EJBObject, Item)

.. EJBObject = fetchBean();
.. sum up (EJBObject)   23

A Remark to Adapters in Component Systems

▲ Component models define *standard*, *unspecific* interfaces

  ▪ E.g., EJBHome / EJBObject

▲ Classes usually define *application-specific* interfaces

▲ To increase reuse of classes, the Adapter pattern(s) can be used to map the application-specific class interfaces to the unspecific component interfaces

▲ Example:

  ▪ In the UNIX shell, all components obey to the pipe-filter interfaces *stdin, stdout, stderr* (untyped channels or streams of bytes)

  ▪ The functional parts of the components have to be *mapped* by some adapter to the unspecific component interfaces.
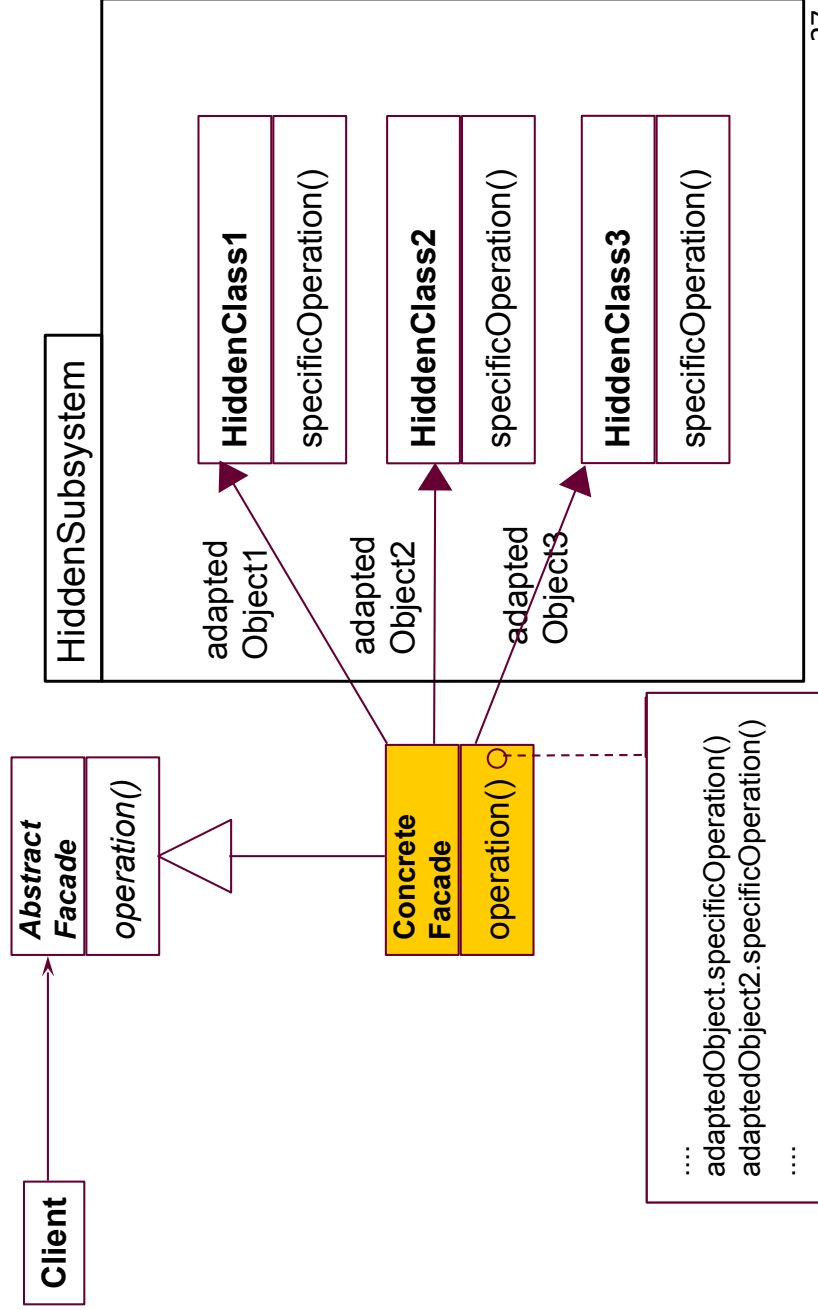
Adapters and Decorators

▲ Similar to a decorator, an adapter inherits its interface from the goal class
  ▪ but adapts the interface

▲ Hence, adapters can be *inserted* into inheritance hierarchies later on

New Extensions

Library

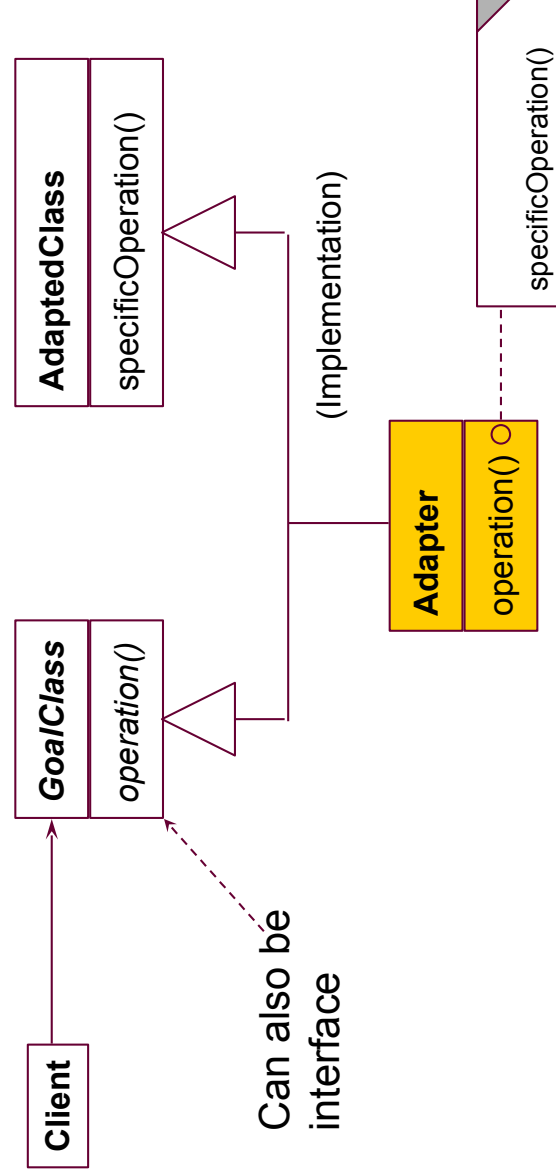Adapter with New Features

Adapted Class

Library
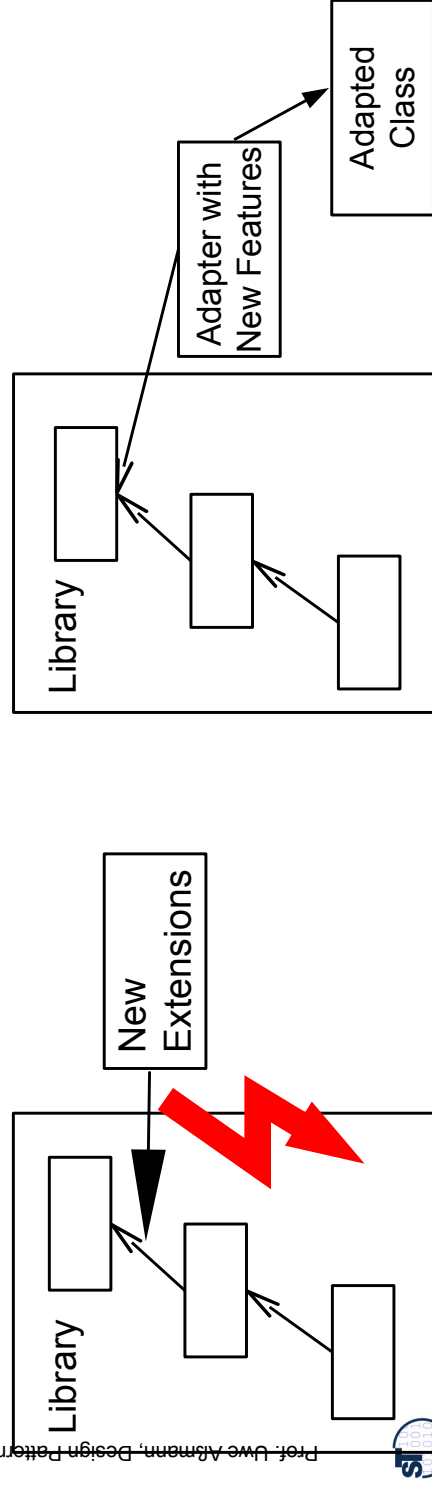
5.3 Facade

▪ A **facade** is an object adapter that hides a complete set of objects (subsystem)

  • Or: a proxy that hides a subsystem

  • The facade has to map its own interface to the interfaces of the hidden objects
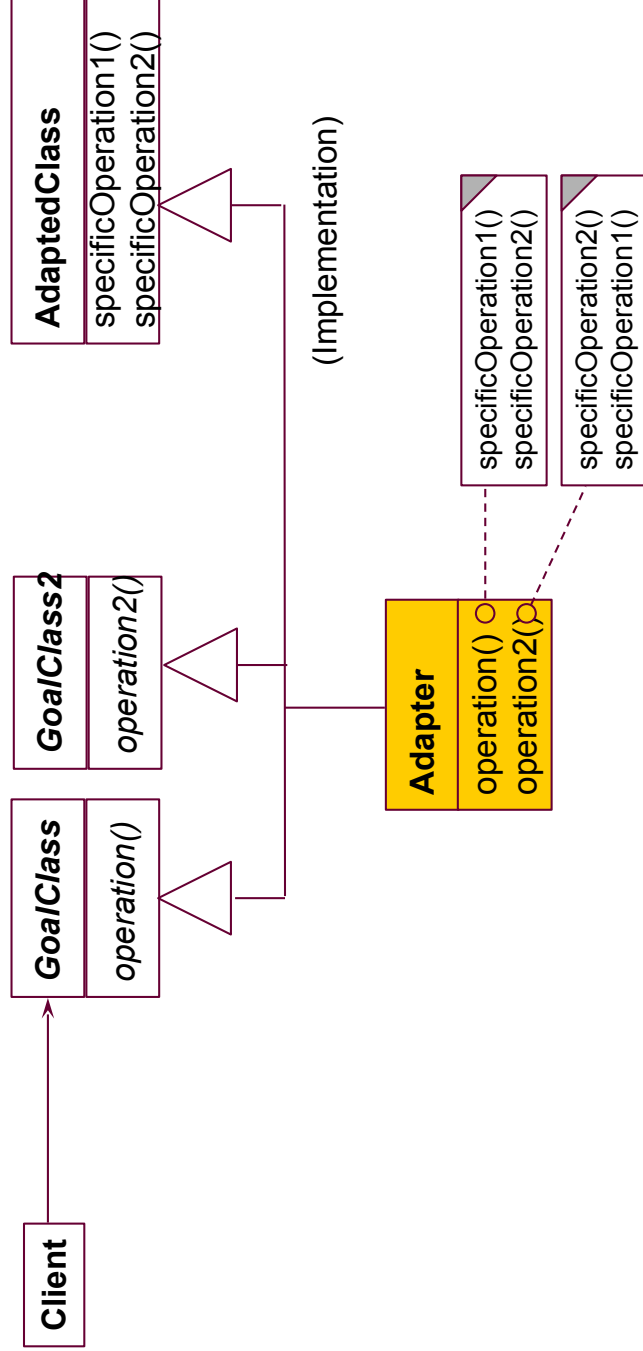
Facade Hides a Subsystem

Client

Abstract Facade
operation()

Concrete Facade
operation()

HiddenSubsystem

HiddenClass1
specificOperation()

HiddenClass2
specificOperation()

HiddenClass3
specificOperation()

adapted Object1
adapted Object2
adapted Object3

...
adaptedObject.specificOperation()
adaptedObject2.specificOperation()
...

5.4 Class Adapter (Integrated Adapter)

- Instead of delegation, class adapters use multiple inheritance

▲

Client

GoalClass
operation()

AdaptedClass
specificOperation()

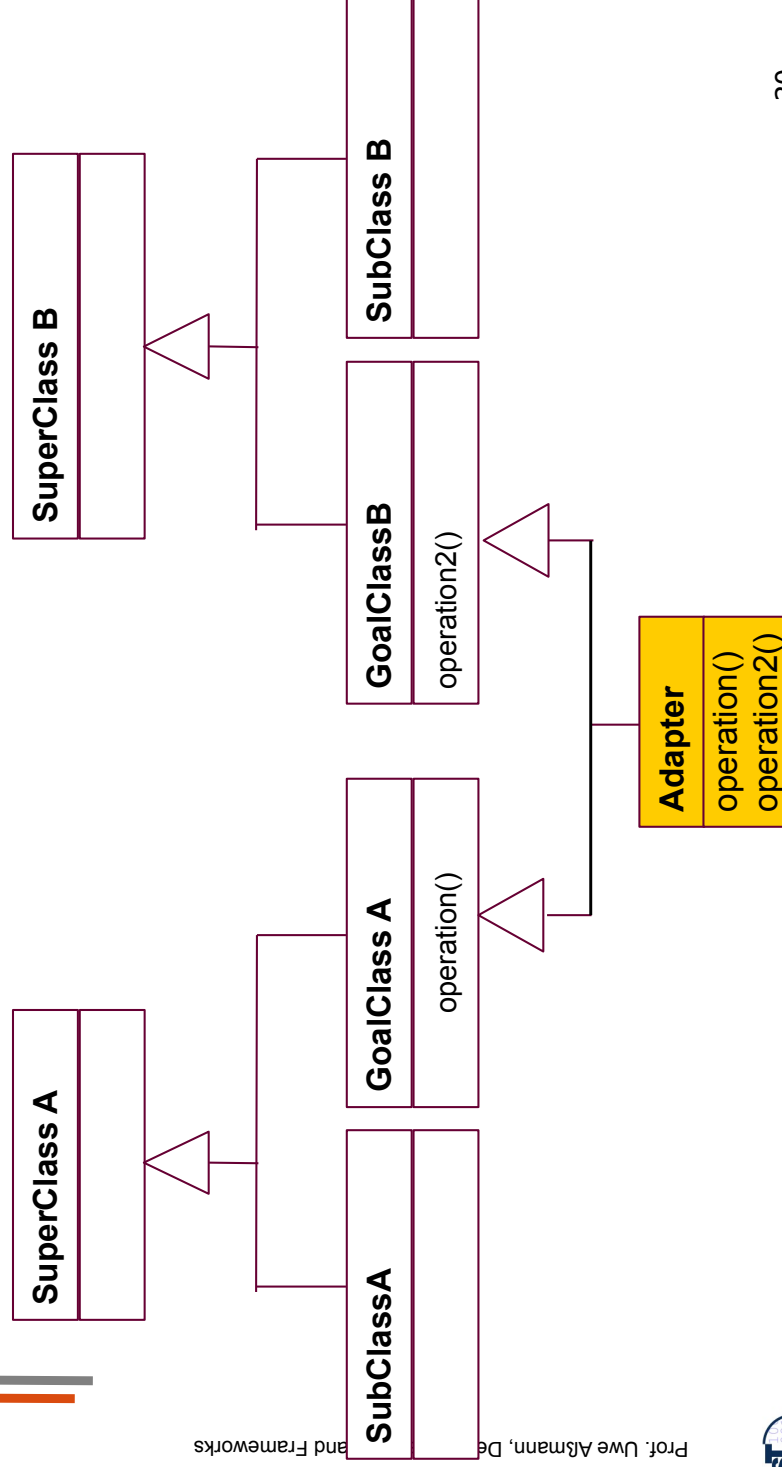(Implementation)

Adapter
operation()
specificOperation()

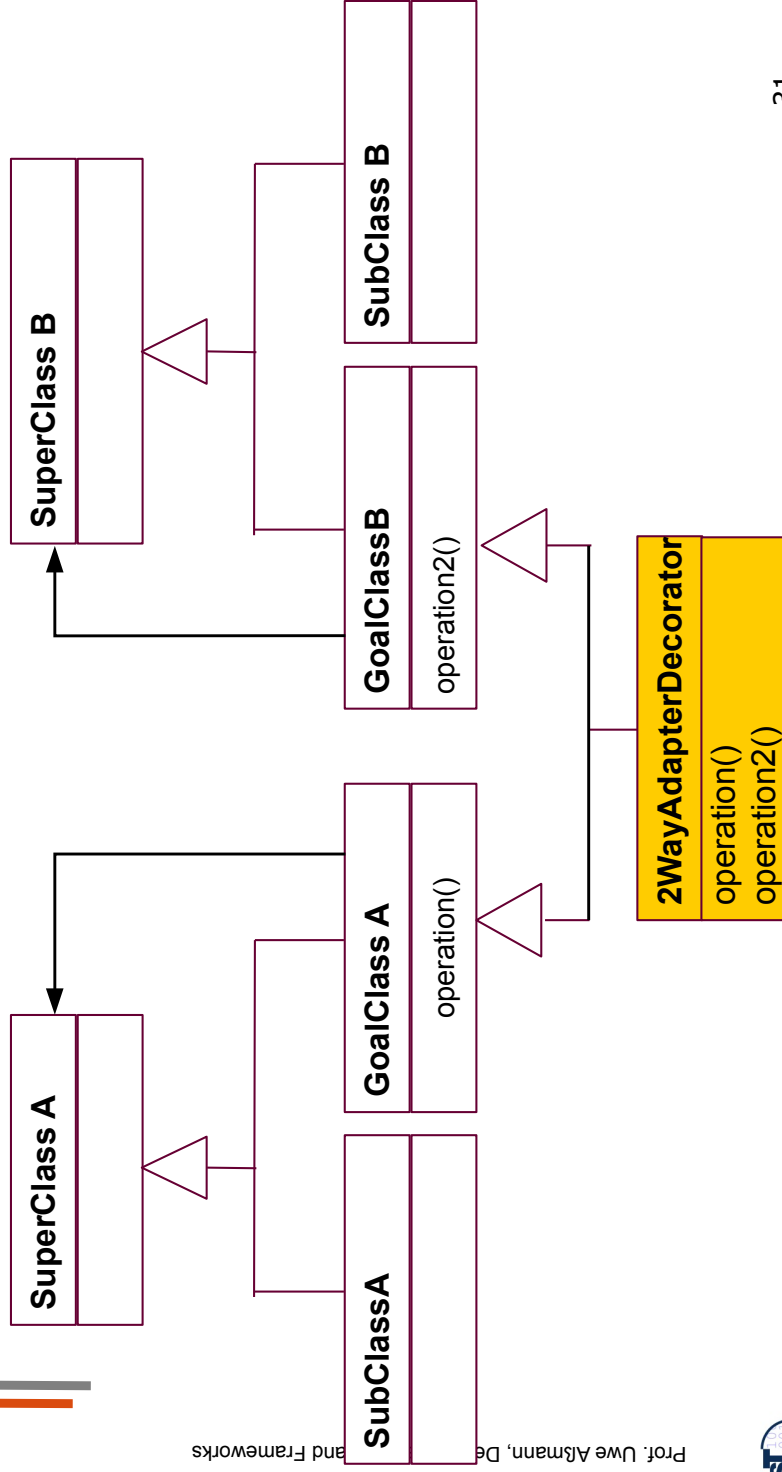Can also be interface

2-Way Class Adapter (Role Mediator)

More than one goal class may exist.
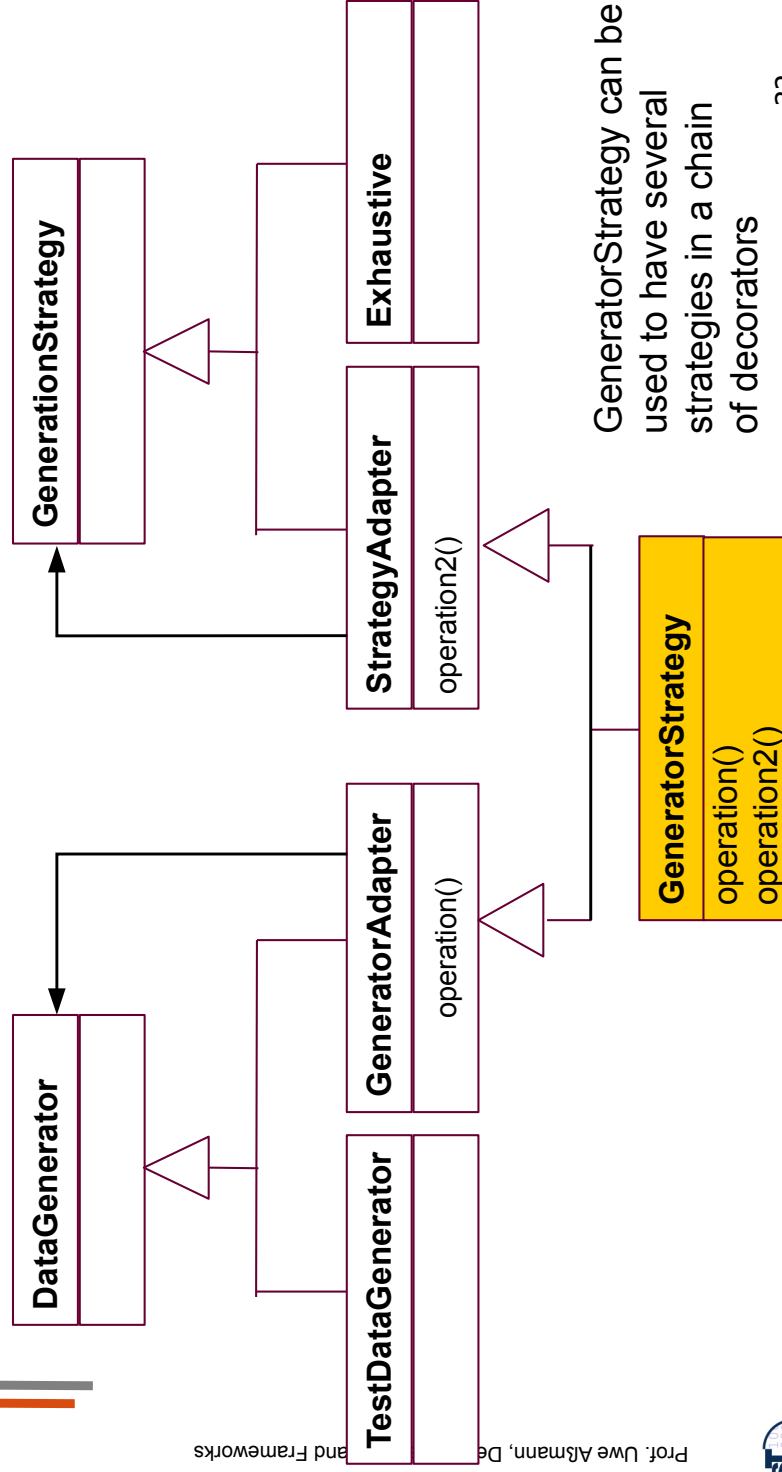Every goal class plays a *role* of the concrete object (see later).

2-Way Adapter for Coupling of Class Hierarchies

2-Way Decorator and Adapter for Coupling of Class Hierarchies

SuperClass A

SuperClass B

GoalClass A
operation()

GoalClassB
operation2()

SubClassA

SubClass B

2WayAdapterDecorator
operation()
operation2()

Ex.: 2-Way Decorator and Adapter for Coupling of Class Hierarchies

GenerationStrategy

DataGenerator

StrategyAdapter
operation2()

GeneratorAdapter
operation()

Exhaustive

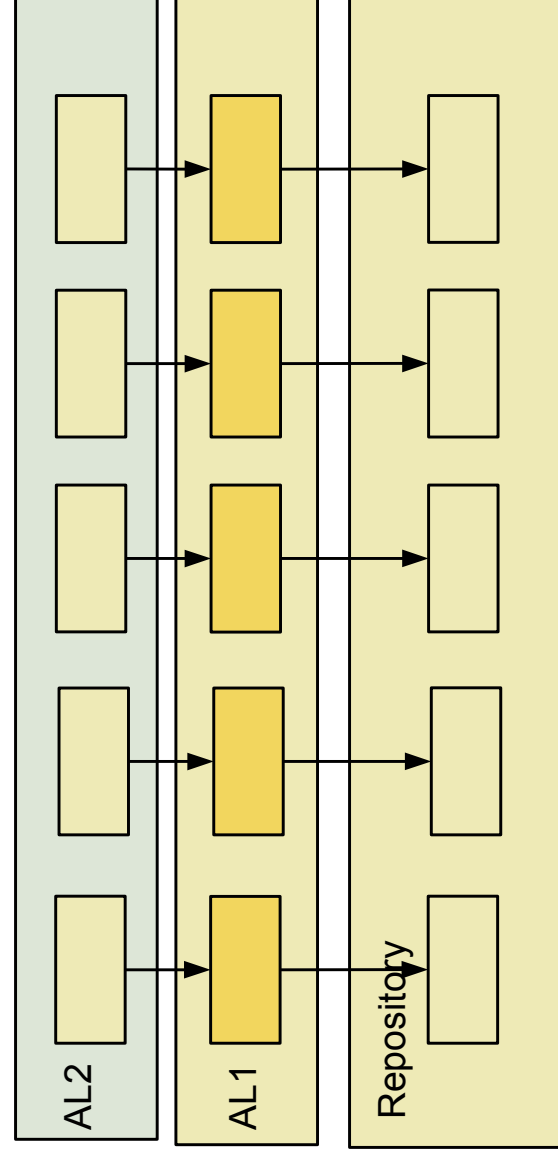TestDataGenerator

GeneratorStrategy
operation()
operation2()

GeneratorStrategy can be used to have several strategies in a chain of decorators

5.5 Adapter Layers

Adapter Layer

- An **Adapter Layer** is a set of adapters hiding a sublayer
  - Every layer has different interfaces (services) that are mapped

AL2

AL1

Repository

Object Skin Layers

▲ An **Object Skin Layer** is a stack of adapter layers in which the adapters vertically form a subject (complex object)

■ Every layer has different interfaces (services) that are mapped, but within the object
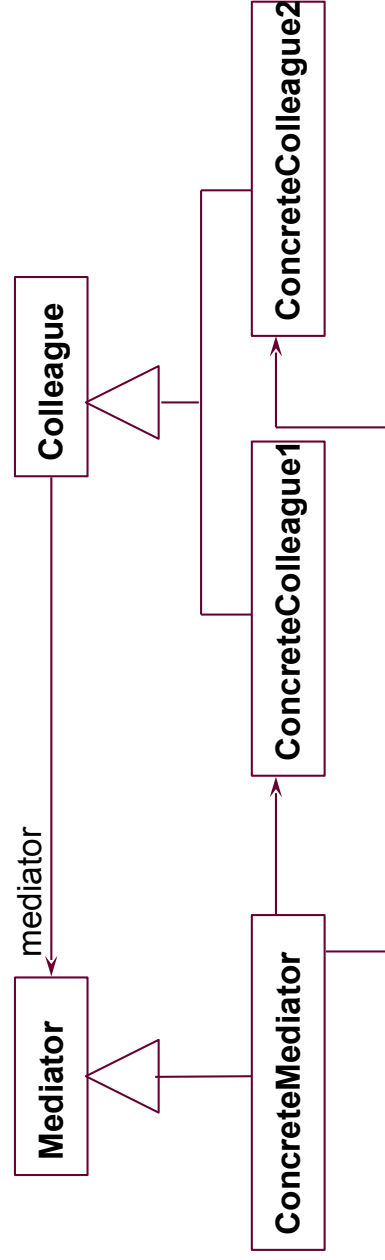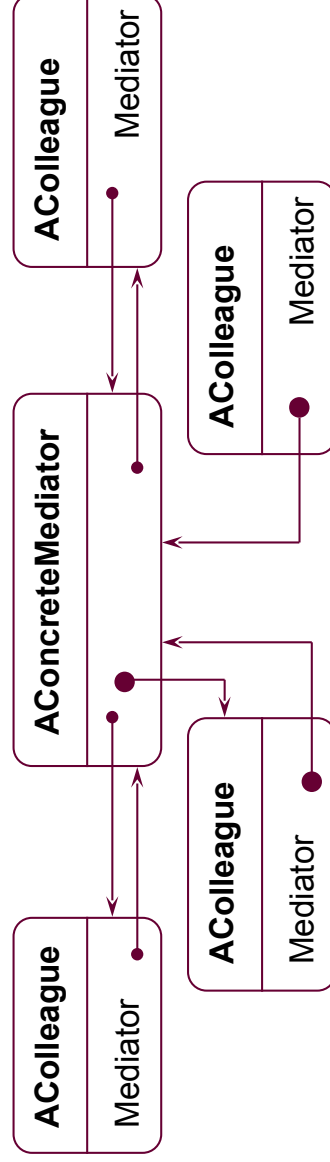

AL2

AL1

Repository
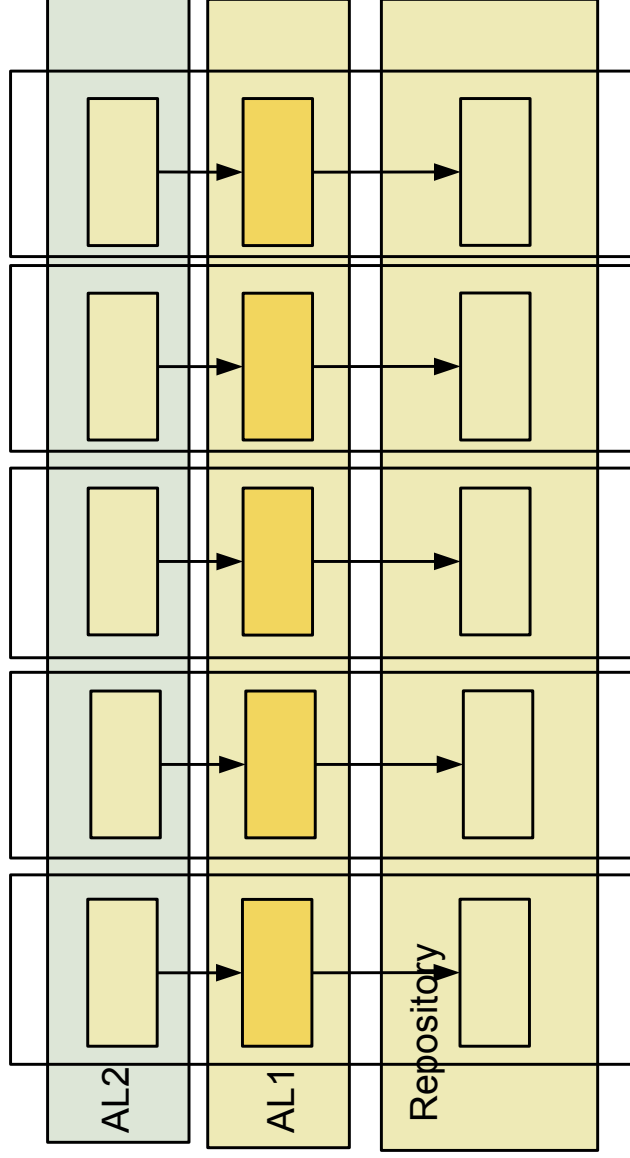
5.6 Mediator (Broker)

Mediator (Broker)

- A mediator is an n-way proxy for communication
  - Combined with a Bridge
- A mediator serves for
  - *Anonymous* communication
  - *Dynamic* communication nets
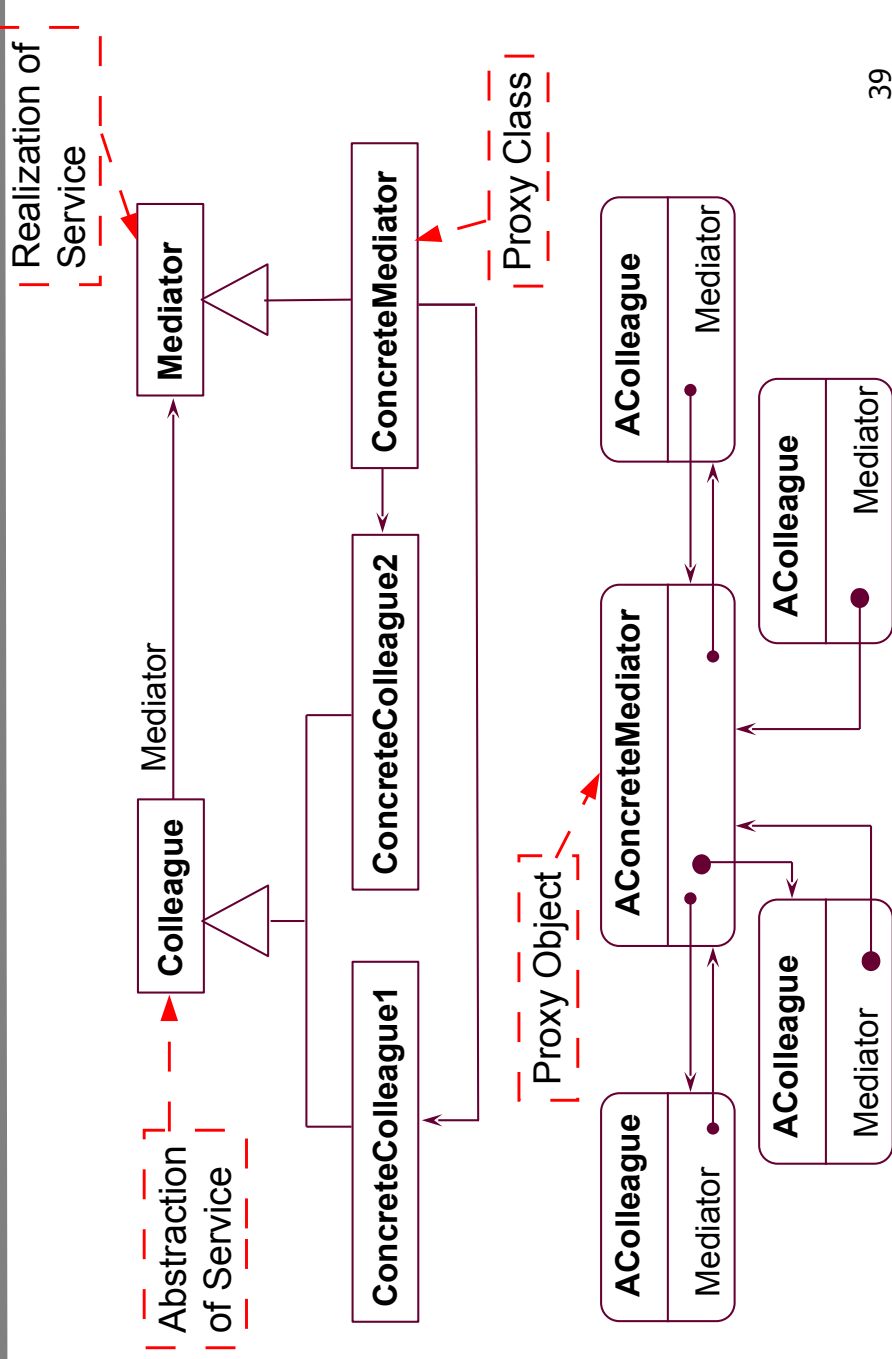

Mediator

Colleague

ConcreteColleague1

ConcreteColleague2

Mediator

mediator

ConcreteMediator

Typical Object Structure:

AColleague — Mediator

AConcreteMediator

AColleague — Mediator

AColleague — Mediator

AColleague — Mediator

Mediator As n-Proxy and Bridge

Realization of Service

Proxy Class

Mediator

Colleague

ConcreteMediator

ConcreteColleague2

ConcreteColleague1

Abstraction of Service

Proxy Object

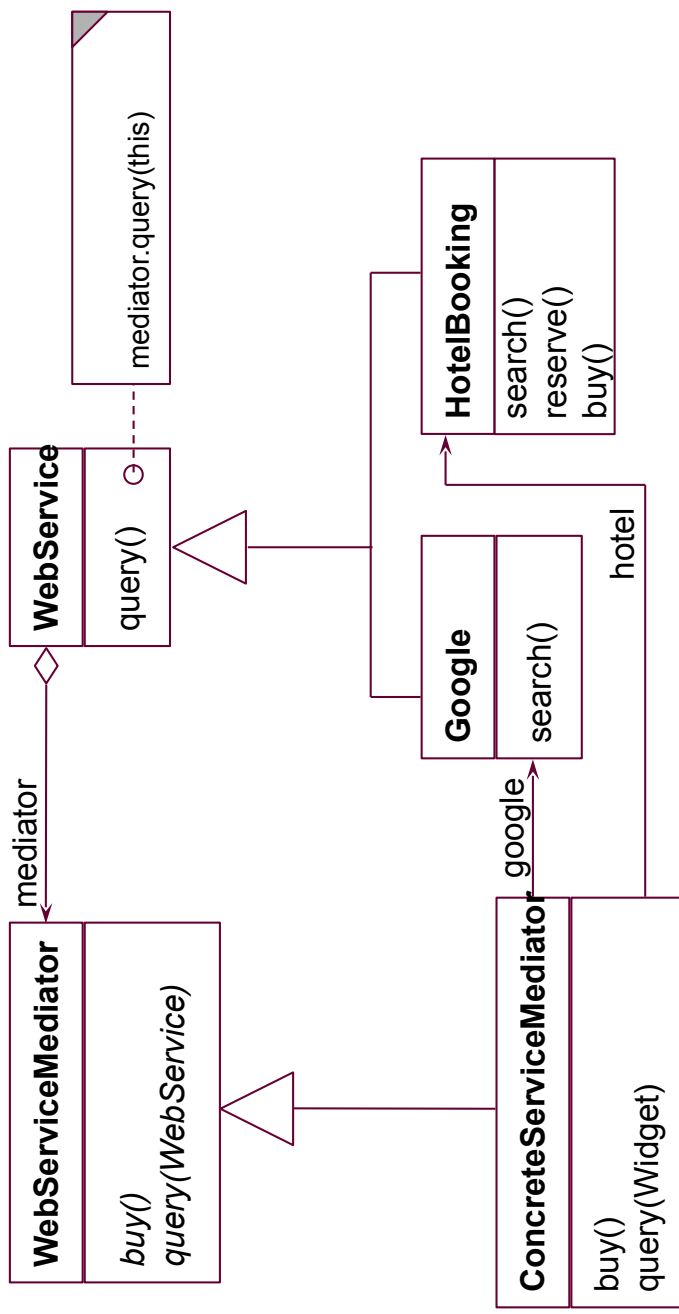ACColleague / Mediator

AConcreteMediator

Intent of Mediator

▲ Proxy object hides all communication partners
  - ■ Every partner uses the mediator object as proxy
  - ■ Clear: real partner is hidden

▲ Bridge links both communication partners
  - ■ Both mediator and partner hierarchies can be varied

▲ ObserverWithChangeManager combines Observer with Mediator

Web Service Brokers

▲ Communication between Web services can be mediated via a broker object (aka object request broker, ORB)

WebServiceMediator

buy()
query(WebService)

ConcreteServiceMediator

buy()
query(Widget)

mediator

WebService

query()

mediator.query(this)

Google

search()

HotelBooking

search()
reserve()
buy()

google

hotel

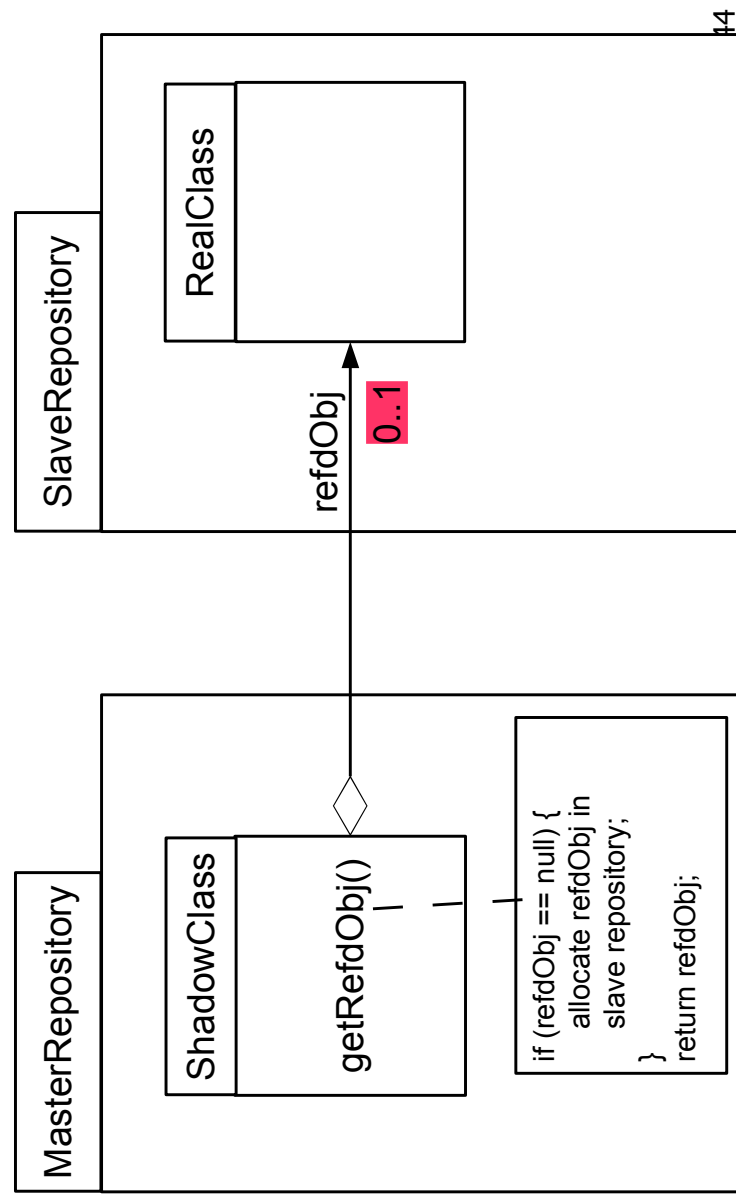
5.7 Coupling Tools with the Repository Connector Pattern

A recent answer…

Coupling of Tools via Repositories

▲ How can two tools collaborate that did not know of each other?

▲ Answer: by coupling their repositories
  - Choose a master and a slave tool
  - Choose a master repository
  - Shadow the master repository in the slave repository

▲ Consequence: all data lies in slave repository, and can be worked on by slave *and* master

---

Coupling of Repositories with "RepositoryConnector"

▲ [Stölzel 2005] connects two repositories of tools with lazy indirection proxies

SlaveRepository

RealClass

refdObj 0..1

MasterRepository

ShadowClass

getRefdObj()

```
if (refdObj == null) {
    allocate refdObj in
    slave repository;
}
return refdObj;
```

Coupling of Repositories with "RepositoryConnector"

- ▲ On demand, objects of *real classes* in the master repository are created in the slave repository
- ▲ Service demands on the master repository are always delegated to the slave repository

SlaveRepository

a':RealA

refdObj

b':RealB

MasterRepository

a:ShadowA
getRefdObj()

b:ShadowB
getRefdObj()
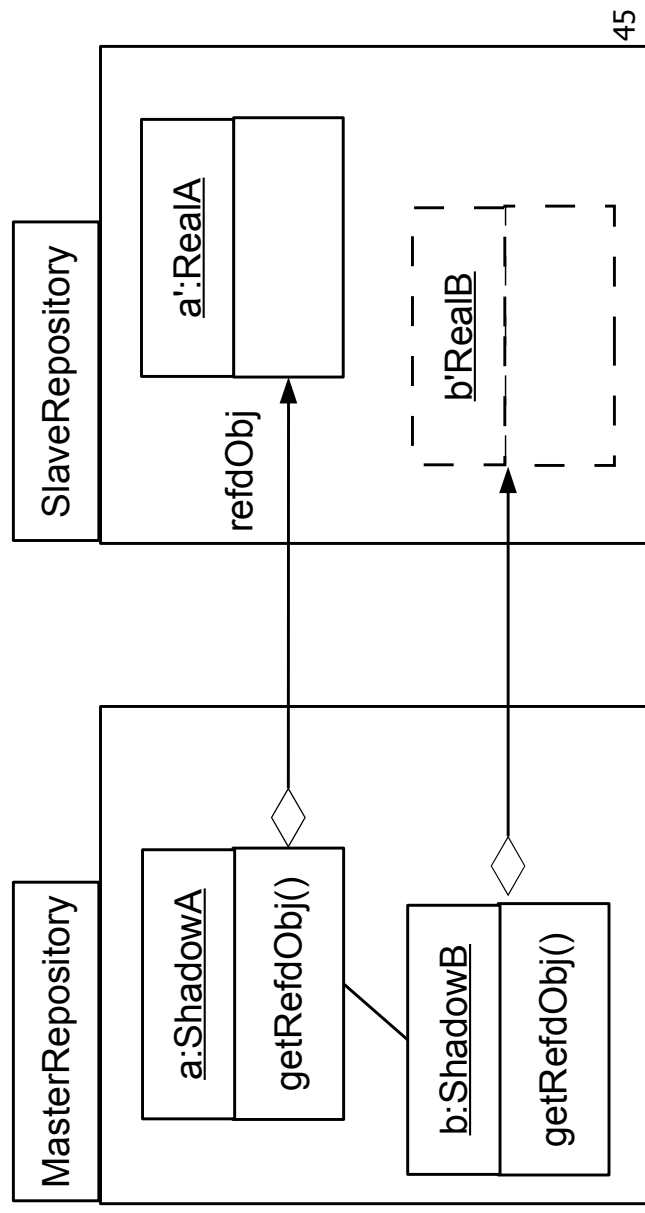
Summary

- ▲ Architectural mismatch between components and tools consists of different assumptions about components, connections, architecture, and building procedure
- ▲ Design patterns, such as extensibility patterns or communication patterns, can bridge architectural mismatches
  - ▪ Data mismatch
  - ▪ Interface mismatch
  - ▪ Protocol mismatch
- ▲ Coupling two tools that had not been foreseen for each other is possible with lazy indirection proxies (RepositoryConnector)
- ▲ With Glue Patterns, reuse of COTS becomes much better

The End